Alibaba Group
阿里巴巴集团

码出高效

阿里巴巴Java开发手册 终极版（1.3.0）

Alibaba Java Coding Guidelines

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

前言

《阿里巴巴 Java 开发手册》是阿里巴巴集团技术团队的集体智慧结晶和经验总结，经历了多次大规模一线实战的检验及不断的完善，系统化地整理成册，反馈给广大开发者。现代软件行业的高速发展对开发者的综合素质要求越来越高，因为不仅是编程知识点，其它维度的知识点也会影响到软件的最终交付质量。比如：数据库的表结构和索引设计缺陷可能带来软件上的架构缺陷或性能风险；工程结构混乱导致后续维护艰难；没有鉴权的漏洞代码易被黑客攻击等等。所以本手册以 Java 开发者为中心视角，划分为编程规约、异常日志、单元测试、安全规约、工程结构、MySQL 数据库六个维度，再根据内容特征，细分成若干二级子目录。根据约束力强弱及故障敏感性，规约依次分为强制、推荐、参考三大类。对于规约条目的延伸信息中，“说明”对内容做了适当扩展和解释；“正例”提倡什么样的编码和实现方式；“反例”说明需要提防的雷区，以及真实的错误案例。

本手册的愿景是**码出高效，码出质量**。现代软件架构都需要协同开发完成，高效协作即降低协同成本，提升沟通效率，所谓无规矩不成方圆，无规范不能协作。众所周知，制订交通法规表面上是要限制行车权，实际上是保障公众的人身安全。试想如果没有限速，没有红绿灯，谁还敢上路行驶。对软件来说，适当的规范和标准绝不是消灭代码内容的创造性、优雅性，而是限制过度个性化，以一种普遍认可的统一方式一起做事，提升协作效率。代码的字里行间流淌的是软件生命中的血液，质量的提升是尽可能少踩坑，杜绝踩重复的坑，切实提升质量意识。

考虑到可以零距离地与众多开发同学进行互动，决定在线维护《手册》内容，此版本号为 `1.3.0` 的 PDF 版本，是对外释放的终极版；其次，我们会在 2017 年 10 月 14 日杭州云栖大会上，进行阿里巴巴 Java 开发规约插件全球首发，插件点此下载，阿里巴巴云效（一站式企业协同研发云）也会集成代码规约扫描引擎。最后，《**码出高效**——阿里巴巴 Java 开发手册详解》即将出版，敬请关注。

目录

前言
一、编程规约 1
（一） 命名风格 1
（二） 常量定义 3
（三） 代码格式 4
（四） OOP 规约 6
（五） 集合处理 9
（六） 并发处理 12
（七） 控制语句 14
（八） 注释规约 16
（九） 其它 17
二、异常日志 18
（一） 异常处理 18
（二） 日志规约 19
三、单元测试 21
四、安全规约 23
五、MySQL 数据库 24
（一） 建表规约 24
（二） 索引规约 25
（三） SQL 语句 27
（四） ORM 映射 28
六、工程结构 30
（一） 应用分层 30
（二） 二方库依赖 31
（三） 服务器 32
附 1：版本历史 34
附 2：本手册专有名词 35

（注：浏览时请使用 PDF 左侧导航栏）

Java 开发手册

版本号	制定团队	更新日期	备注
1.3.0	阿里巴巴集团技术团队	**2017.9.25**	增加单元测试规约（PDF 终极版）

一、编程规约

（一）命名风格

1. 【强制】代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。
   反例：_name / __name / $Object / name_ / name$ / Object$

2. 【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。
   说明：正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注意，即使纯拼音命名方式也要避免采用。
   正例：alibaba / taobao / youku / hangzhou 等国际通用的名称，可视同英文。
   反例：DaZhePromotion [打折] / getPingfenByName() [评分] / int 某变量 = 3

3. 【强制】类名使用 UpperCamelCase 风格，必须遵从驼峰形式，但以下情形例外：DO / BO / DTO / VO / AO
   正例：MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion
   反例：macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

4. 【强制】方法名、参数名、成员变量、局部变量都统一使用 lowerCamelCase 风格，必须遵从驼峰形式。
   正例： localValue / getHttpMessage() / inputUserId

5. 【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。
   正例：MAX_STOCK_COUNT
   反例：MAX_COUNT

6. 【强制】抽象类命名使用 Abstract 或 Base 开头；异常类命名使用 Exception 结尾；测试类命名以它要测试的类的名称开始，以 Test 结尾。

7. 【强制】中括号是数组类型的一部分，数组定义如下：String[] args;
   反例：使用 String args[]的方式来定义。

8. 【强制】POJO 类中布尔类型的变量，都不要加 is，否则部分框架解析会引起序列化错误。
   反例：定义为基本数据类型 Boolean isDeleted；的属性，它的方法也是 isDeleted()，RPC

框架在反向解析的时候，“以为”对应的属性名称是 deleted，导致属性获取不到，进而抛出异常。

9. 【强制】包名统一使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名统一使用单数形式，但是类名如果有复数含义，类名可以使用复数形式。

正例： 应用工具类包名为 com.alibaba.open.util、类名为 MessageUtils（此规则参考 spring 的框架结构）

10. 【强制】杜绝完全不规范的缩写，避免望文不知义。

反例：AbstractClass“缩写”命名成 AbsClass；condition“缩写”命名成 condi，此类随意缩写严重降低了代码的可阅读性。

11. 【推荐】为了达到代码自解释的目标，任何自定义编程元素在命名时，使用尽量完整的单词组合来表达其意。

正例：从远程仓库拉取代码的类命名为 PullCodeFromRemoteRepository。

反例：变量 int a; 的随意命名方式。

12. 【推荐】如果模块、接口、类、方法使用了设计模式，在命名时体现出具体模式。

说明：将设计模式体现在名字中，有利于阅读者快速理解架构设计理念。

正例：
```
public class OrderFactory;
public class LoginProxy;
public class ResourceObserver;
```

13. 【推荐】接口类中的方法和属性不要加任何修饰符号（public 也不要加），保持代码的简洁性，并加上有效的 Javadoc 注释。尽量不要在接口里定义变量，如果一定要定义变量，肯定是与接口方法相关，并且是整个应用的基础常量。

正例：接口方法签名：void f();

接口基础常量表示：String COMPANY = "alibaba";

反例：接口方法定义：public abstract void f();

说明：JDK8 中接口允许有默认实现，那么这个 default 方法，是对所有实现类都有价值的默认实现。

14. 接口和实现类的命名有两套规则：

1）【强制】对于 Service 和 DAO 类，基于 SOA 的理念，暴露出来的服务一定是接口，内部的实现类用 Impl 的后缀与接口区别。

正例：CacheServiceImpl 实现 CacheService 接口。

2）【推荐】如果是形容能力的接口名称，取对应的形容词做接口名（通常是–able 的形式）。

正例：AbstractTranslator 实现 Translatable。

15．【参考】枚举类名建议带上 Enum 后缀，枚举成员名称需要全大写，单词间用下划线隔开。

说明：枚举其实就是特殊的常量类，且构造方法被默认强制是私有。

正例：枚举名字为 ProcessStatusEnum 的成员名称：SUCCESS / UNKOWN_REASON。

16．【参考】各层命名规约：

A）Service/DAO 层方法命名规约

1） 获取单个对象的方法用 get 做前缀。

2） 获取多个对象的方法用 list 做前缀。

3） 获取统计值的方法用 count 做前缀。

4） 插入的方法用 save/insert 做前缀。

5） 删除的方法用 remove/delete 做前缀。

6） 修改的方法用 update 做前缀。

B）领域模型命名规约

1） 数据对象：xxxDO，xxx 即为数据表名。

2） 数据传输对象：xxxDTO，xxx 为业务领域相关的名称。

3） 展示对象：xxxVO，xxx 一般为网页名称。

4） POJO 是 DO/DTO/BO/VO 的统称，禁止命名成 xxxPOJO。

（二）常量定义

1．【强制】不允许任何魔法值（即未经定义的常量）直接出现在代码中。

反例：
```
String key = "Id#taobao_" + tradeId;
cache.put(key, value);
```

2．【强制】long 或者 Long 初始赋值时，使用大写的 L，不能是小写的 l，小写容易跟数字 1 混淆，造成误解。

说明：Long a = 2l; 写的是数字的 21，还是 Long 型的 2?

3．【推荐】不要使用一个常量类维护所有常量，按常量功能进行归类，分开维护。

说明：大而全的常量类，非得使用查找功能才能定位到修改的常量，不利于理解和维护。

正例：缓存相关常量放在类 CacheConsts 下；系统配置相关常量放在类 ConfigConsts 下。

4．【推荐】常量的复用层次有五层：跨应用共享常量、应用内共享常量、子工程内共享常量、包内共享常量、类内共享常量。

1） 跨应用共享常量：放置在二方库中，通常是 client.jar 中的 constant 目录下。

2） 应用内共享常量：放置在一方库中，通常是 modules 中的 constant 目录下。

反例：易懂变量也要统一定义成应用内共享常量，两位攻城师在两个类中分别定义了表示“是”的变量：

类 A 中：public static final String YES = "yes";

类 B 中：public static final String YES = "y";

A.YES.equals(B.YES)，预期是 true，但实际返回为 false，导致线上问题。

3） 子工程内部共享常量：即在当前子工程的 constant 目录下。

4） 包内共享常量：即在当前包下单独的 constant 目录下。

5） 类内共享常量：直接在类内部 private static final 定义。

5. 【推荐】如果变量值仅在一个范围内变化，且带有名称之外的延伸属性，定义为枚举类。下面正例中的数字就是延伸信息，表示星期几。

正例：public Enum { MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);}

（三）代码格式

1. 【强制】大括号的使用约定。如果是大括号内为空，则简洁地写成{}即可，不需要换行；如果是非空代码块则：

1） 左大括号前不换行。

2） 左大括号后换行。

3） 右大括号前换行。

4） 右大括号后还有 else 等代码则不换行；表示终止的右大括号后必须换行。

2. 【强制】 左小括号和字符之间不出现空格；同样，右小括号和字符之间也不出现空格。详见第 5 条下方正例提示。

反例：if (空格 a == b 空格)

3. 【强制】if/for/while/switch/do 等保留字与括号之间都必须加空格。

4. 【强制】任何二目、三目运算符的左右两边都需要加一个空格。

说明：运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号等。

5. 【强制】采用 4 个空格缩进，禁止使用 tab 字符。

说明：如果使用 tab 缩进，必须设置 1 个 tab 为 4 个空格。IDEA 设置 tab 为 4 个空格时，请勿勾选 Use tab character；而在 eclipse 中，必须勾选 insert spaces for tabs。

正例： （涉及 1-5 点）

```
public static void main(String[] args) {
    // 缩进 4 个空格
    String say = "hello";
    // 运算符的左右必须有一个空格
    int flag = 0;
```

```
        // 关键词 if 与括号之间必须有一个空格，括号内的 f 与左括号，0 与右括号不需要空格
        if (flag == 0) {
            System.out.println(say);
        }

        // 左大括号前加空格且不换行；左大括号后换行
        if (flag == 1) {
            System.out.println("world");
        // 右大括号前换行，右大括号后有 else，不用换行
        } else {
            System.out.println("ok");
        // 在右大括号后直接结束，则必须换行
        }
    }
```

6．【强制】注释的双斜线与注释内容之间有且仅有一个空格。

正例：// 注释内容，注意在//和注释内容之间有一个空格。

7．【强制】单行字符数限制不超过 120 个，超出需要换行，换行时遵循如下原则：

1） 第二行相对第一行缩进 4 个空格，从第三行开始，不再继续缩进，参考示例。

2） 运算符与下文一起换行。

3） 方法调用的点符号与下文一起换行。

4） 方法调用时，多个参数，需要换行时，在逗号后进行。

5） 在括号前不要换行，见反例。

正例：

```
StringBuffer sb = new StringBuffer();
// 超过 120 个字符的情况下，换行缩进 4 个空格，点号和方法名称一起换行
sb.append("zi").append("xin")...
    .append("huang")...
    .append("huang")...
    .append("huang");
```

反例：

```
StringBuffer sb = new StringBuffer();
// 超过 120 个字符的情况下，不要在括号前换行
sb.append("zi").append("xin")...append
    ("huang");

// 参数很多的方法调用可能超过 120 个字符，不要在逗号前换行
 method(args1, args2, args3, ...
    , argsX);
```

8．【强制】方法参数在定义和传入时，多个参数逗号后边必须加空格。

正例：下例中实参的"a",后边必须要有一个空格。

```
method("a", "b", "c");
```

9．【强制】IDE 的 text file encoding 设置为 UTF-8; IDE 中文件的换行符使用 Unix 格式，不要使用 Windows 格式。

10.【推荐】没有必要增加若干空格来使某一行的字符与上一行对应位置的字符对齐。

正例：

```
int a = 3;
long b = 4L;
float c = 5F;
StringBuffer sb = new StringBuffer();
```

说明：增加 sb 这个变量，如果需要对齐，则给 a、b、c 都要增加几个空格，在变量比较多的情况下，是一种累赘的事情。

11.【推荐】方法体内的执行语句组、变量的定义语句组、不同的业务逻辑之间或者不同的语义之间插入一个空行。相同业务逻辑和语义之间不需要插入空行。

说明：没有必要插入多个空行进行隔开。

（四）OOP 规约

1.【强制】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。

2.【强制】所有的覆写方法，必须加@Override 注解。

说明：getObject()与 get0bject()的问题。一个是字母的 O，一个是数字的 0，加@Override 可以准确判断是否覆盖成功。另外，如果在抽象类中对方法签名进行修改，其实现类会马上编译报错。

3.【强制】相同参数类型，相同业务含义，才可以使用 Java 的可变参数，避免使用 Object。

说明：可变参数必须放置在参数列表的最后。（提倡同学们尽量不用可变参数编程）

正例：`public User getUsers(String type, Integer... ids) {...}`

4.【强制】外部正在调用或者二方库依赖的接口，不允许修改方法签名，避免对接口调用方产生影响。接口过时必须加@Deprecated 注解，并清晰地说明采用的新接口或者新服务是什么。

5.【强制】不能使用过时的类或方法。

说明：java.net.URLDecoder 中的方法 decode(String encodeStr) 这个方法已经过时，应该使用双参数 decode(String source, String encode)。接口提供方既然明确是过时接口，那么有义务同时提供新的接口；作为调用方来说，有义务去考证过时方法的新实现是什么。

6.【强制】Object 的 equals 方法容易抛空指针异常，应使用常量或确定有值的对象来调用 equals。

正例：`"test".equals(object);`

反例：`object.equals("test");`

说明：推荐使用 java.util.Objects#equals（JDK7 引入的工具类）

7.【强制】所有的相同类型的包装类对象之间值的比较，全部使用 equals 方法比较。

说明：对于 Integer var = ? 在-128 至 127 范围内的赋值，Integer 对象是在

IntegerCache.cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用==进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

8. 关于基本数据类型与包装数据类型的使用标准如下：

   1） 【强制】所有的 POJO 类属性必须使用包装数据类型。

   2） 【强制】RPC 方法的返回值和参数必须使用包装数据类型。

   3） 【推荐】所有的局部变量使用基本数据类型。

   说明：POJO 类属性没有初值是提醒使用者在需要使用时，必须自己显式地进行赋值，任何 NPE 问题，或者入库检查，都由使用者来保证。

   正例：数据库的查询结果可能是 null，因为自动拆箱，用基本数据类型接收有 NPE 风险。

   反例：比如显示成交总额涨跌情况，即正负 x%，x 为基本数据类型，调用的 RPC 服务，调用不成功时，返回的是默认值，页面显示为 0%，这是不合理的，应该显示成中划线。所以包装数据类型的 null 值，能够表示额外的信息，如：远程调用失败，异常退出。

9. 【强制】定义 DO/DTO/VO 等 POJO 类时，不要设定任何属性默认值。

   反例：POJO 类的 gmtCreate 默认值为 new Date();但是这个属性在数据提取时并没有置入具体值，在更新其它字段时又附带更新了此字段，导致创建时间被修改成当前时间。

10. 【强制】序列化类新增属性时，请不要修改 serialVersionUID 字段，避免反序列失败；如果完全不兼容升级，避免反序列化混乱，那么请修改 serialVersionUID 值。

    说明：注意 serialVersionUID 不一致会抛出序列化运行时异常。

11. 【强制】构造方法里面禁止加入任何业务逻辑，如果有初始化逻辑，请放在 init 方法中。

12. 【强制】POJO 类必须写 toString 方法。使用 IDE 的中工具：source> generate toString 时，如果继承了另一个 POJO 类，注意在前面加一下 super.toString。

    说明：在方法执行抛出异常时，可以直接调用 POJO 的 toString()方法打印其属性值，便于排查问题。

13. 【推荐】使用索引访问用 String 的 split 方法得到的数组时，需做最后一个分隔符后有无内容的检查，否则会有抛 IndexOutOfBoundsException 的风险。

    说明：

    ```
    String str = "a,b,c,,";
    String[] ary = str.split(",");
    // 预期大于 3，结果是 3
    System.out.println(ary.length);
    ```

14. 【推荐】当一个类有多个构造方法，或者多个同名方法，这些方法应该按顺序放置在一起，便于阅读，此条规则优先于第 15 条规则。

15. 【推荐】 类内方法定义顺序依次是：公有方法或保护方法 > 私有方法 > getter/setter 方法。

说明：公有方法是类的调用者和维护者最关心的方法，首屏展示最好；保护方法虽然只是子类关心，也可能是“模板设计模式”下的核心方法；而私有方法外部一般不需要特别关心，是一个黑盒实现；因为承载的信息价值较低，所有 Service 和 DAO 的 getter/setter 方法放在类体最后。

16. 【推荐】setter 方法中，参数名称与类成员变量名称一致，this.成员名 = 参数名。在 getter/setter 方法中，不要增加业务逻辑，增加排查问题的难度。

反例：

```
public Integer getData() {
    if (true) {
        return this.data + 100;
    } else {
        return this.data - 100;
    }
 }
```

17. 【推荐】循环体内，字符串的连接方式，使用 StringBuilder 的 append 方法进行扩展。

说明：反编译出的字节码文件显示每次循环都会 new 出一个 StringBuilder 对象，然后进行 append 操作，最后通过 toString 方法返回 String 对象，造成内存资源浪费。

反例：

```
String str = "start";
for (int i = 0; i < 100; i++) {
    str = str + "hello";
}
```

18. 【推荐】final 可以声明类、成员变量、方法、以及本地变量，下列情况使用 final 关键字：

1） 不允许被继承的类，如：String 类。

2） 不允许修改引用的域对象，如：POJO 类的域变量。

3） 不允许被重写的方法，如：POJO 类的 setter 方法。

4） 不允许运行过程中重新赋值的局部变量。

5） 避免上下文重复使用一个变量，使用 final 描述可以强制重新定义一个变量，方便更好地进行重构。

19. 【推荐】慎用 Object 的 clone 方法来拷贝对象。

说明：对象的 clone 方法默认是浅拷贝，若想实现深拷贝需要重写 clone 方法实现属性对象的拷贝。

20. 【推荐】类成员与方法访问控制从严：

1） 如果不允许外部直接通过 new 来创建对象，那么构造方法必须是 private。

2） 工具类不允许有 public 或 default 构造方法。

3） 类非 static 成员变量并且与子类共享，必须是 protected。

4） 类非 static 成员变量并且仅在本类使用，必须是 private。

5） 类 static 成员变量如果仅在本类使用，必须是 private。

6） 若是 static 成员变量，必须考虑是否为 final。

7） 类成员方法只供类内部调用，必须是 private。

8） 类成员方法只对继承类公开，那么限制为 protected。

说明：任何类、方法、参数、变量，严控访问范围。过于宽泛的访问范围，不利于模块解耦。思考：如果是一个 private 的方法，想删除就删除，可是一个 public 的 service 方法，或者一个 public 的成员变量，删除一下，不得手心冒点汗吗？变量像自己的小孩，尽量在自己的视线内，变量作用域太大，无限制的到处跑，那么你会担心的。

（五）集合处理

1. 【强制】关于 hashCode 和 equals 的处理，遵循如下规则：

   1） 只要重写 equals，就必须重写 hashCode。

   2） 因为 Set 存储的是不重复的对象，依据 hashCode 和 equals 进行判断，所以 Set 存储的对象必须重写这两个方法。

   3） 如果自定义对象做为 Map 的键，那么必须重写 hashCode 和 equals。

   说明：String 重写了 hashCode 和 equals 方法，所以我们可以非常愉快地使用 String 对象作为 key 来使用。

2. 【强制】ArrayList 的 subList 结果不可强转成 ArrayList，否则会抛出 ClassCastException 异常，即 java.util.RandomAccessSubList cannot be cast to java.util.ArrayList.

   说明：subList 返回的是 ArrayList 的内部类 SubList，并不是 ArrayList ，而是 ArrayList 的一个视图，对于 SubList 子列表的所有操作最终会反映到原列表上。

3. 【强制】在 subList 场景中，高度注意对原集合元素个数的修改，会导致子列表的遍历、增加、删除均会产生 ConcurrentModificationException 异常。

4. 【强制】使用集合转数组的方法，必须使用集合的 toArray(T[] array)，传入的是类型完全一样的数组，大小就是 list.size()。

   说明：使用 toArray 带参方法，入参分配的数组空间不够大时，toArray 方法内部将重新分配内存空间，并返回新数组地址；如果数组元素大于实际所需，下标为[list.size()]的数组元素将被置为 null，其它数组元素保持原值，因此最好将方法入参数组大小定义与集合元素个数一致。

   正例：

```
List<String> list = new ArrayList<String>(2);
list.add("guan");
list.add("bao");
String[] array = new String[list.size()];
array = list.toArray(array);
```

反例：直接使用 toArray 无参方法存在问题，此方法返回值只能是 Object[]类，若强转其它类型数组将出现 ClassCastException 错误。

5．【强制】使用工具类 Arrays.asList()把数组转换成集合时，不能使用其修改集合相关的方法，它的 add/remove/clear 方法会抛出 UnsupportedOperationException 异常。

说明：asList 的返回对象是一个 Arrays 内部类，并没有实现集合的修改方法。Arrays.asList 体现的是适配器模式，只是转换接口，后台的数据仍是数组。

```
String[] str = new String[] { "you", "wu" };

List list = Arrays.asList(str);
```

第一种情况：list.add("yangguanbao"); 运行时异常。

第二种情况：str[0] = "gujin"; 那么 list.get(0)也会随之修改。

6．【强制】泛型通配符<? extends T>来接收返回的数据，此写法的泛型集合不能使用 add 方法，而<? super T>不能使用 get 方法，做为接口调用赋值时易出错。

说明：扩展说一下 PECS(Producer Extends Consumer Super)原则：第一、频繁往外读取内容的，适合用<? extends T>。第二、经常往里插入的，适合用<? super T>。

7．【强制】不要在 foreach 循环里进行元素的 remove/add 操作。remove 元素请使用 Iterator 方式，如果并发操作，需要对 Iterator 对象加锁。

正例：

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String item = iterator.next();
    if (删除元素的条件) {
        iterator.remove();
    }
}
```

反例：

```
List<String> list = new ArrayList<String>();
list.add("1");
list.add("2");
for (String item : list) {
    if ("1".equals(item)) {
        list.remove(item);
    }
}
```

说明：以上代码的执行结果肯定会出乎大家的意料，那么试一下把“1”换成“2”，会是同样的结果吗？

8．【强制】 在 JDK7 版本及以上，Comparator 要满足如下三个条件，不然 Arrays.sort，Collections.sort 会报 IllegalArgumentException 异常。

说明：三个条件如下

1） x，y 的比较结果和 y，x 的比较结果相反。

2） x>y，y>z，则 x>z。

3） x=y，则 x，z 比较结果和 y，z 比较结果相同。

反例：下例中没有处理相等的情况，实际使用中可能会出现异常：

```
new Comparator<Student>() {
    @Override
    public int compare(Student o1, Student o2) {
        return o1.getId() > o2.getId() ? 1 : -1;
    }
};
```

9. 【推荐】集合初始化时，指定集合初始值大小。

说明：HashMap 使用 HashMap(int initialCapacity) 初始化，

正例：initialCapacity = (需要存储的元素个数 / 负载因子) + 1。注意负载因子(即 loader factor）默认为 0.75，如果暂时无法确定初始值大小，请设置为 16（即默认值）。

反例：HashMap 需要放置 1024 个元素，由于没有设置容量初始大小，随着元素不断增加，容量 7 次被迫扩大，resize 需要重建 hash 表，严重影响性能。

10. 【推荐】使用 entrySet 遍历 Map 类集合 KV，而不是 keySet 方式进行遍历。

说明：keySet 其实是遍历了 2 次，一次是转为 Iterator 对象，另一次是从 hashMap 中取出 key 所对应的 value。而 entrySet 只是遍历了一次就把 key 和 value 都放到了 entry 中，效率更高。如果是 JDK8，使用 Map.foreach 方法。

正例：values()返回的是 V 值集合，是一个 list 集合对象；keySet()返回的是 K 值集合，是一个 Set 集合对象；entrySet()返回的是 K-V 值组合集合。

11. 【推荐】高度注意 Map 类集合 K/V 能不能存储 null 值的情况，如下表格：

集合类	Key	Value	Super	说明
Hashtable	不允许为 null	不允许为 null	Dictionary	线程安全
ConcurrentHashMap	不允许为 null	不允许为 null	AbstractMap	锁分段技术（JDK8:CAS）
TreeMap	不允许为 null	允许为 null	AbstractMap	线程不安全
HashMap	允许为 null	允许为 null	AbstractMap	线程不安全

反例： 由于 HashMap 的干扰，很多人认为 ConcurrentHashMap 是可以置入 null 值，而事实上，存储 null 值时会抛出 NPE 异常。

12. 【参考】合理利用好集合的有序性(sort)和稳定性(order)，避免集合的无序性(unsort)和不稳定性(unorder)带来的负面影响。

说明：有序性是指遍历的结果是按某种比较规则依次排列的。稳定性指集合每次遍历的元素次序是一定的。如：ArrayList 是 order/unsort；HashMap 是 unorder/unsort；TreeSet 是 order/sort。

13．【参考】利用 Set 元素唯一的特性，可以快速对一个集合进行去重操作，避免使用 List 的 contains 方法进行遍历、对比、去重操作。

（六）并发处理

1．【强制】获取单例对象需要保证线程安全，其中的方法也要保证线程安全。

说明：资源驱动类、工具类、单例工厂类都需要注意。

2．【强制】创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。

正例：

```
public class TimerTaskThread extends Thread {
public TimerTaskThread() {
    super.setName("TimerTaskThread");
    ...
}
```

3．【强制】线程资源必须通过线程池提供，不允许在应用中自行显式创建线程。

说明：使用线程池的好处是减少在创建和销毁线程上所花的时间以及系统资源的开销，解决资源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题。

4．【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors 返回的线程池对象的弊端如下：

1）FixedThreadPool 和 SingleThreadPool:

允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。

2）CachedThreadPool 和 ScheduledThreadPool:

允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

5．【强制】SimpleDateFormat 是线程不安全的类，一般不要定义为 static 变量，如果定义为 static，必须加锁，或者使用 DateUtils 工具类。

正例：注意线程安全，使用 DateUtils。亦推荐如下处理：

```
private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>() {
    @Override
    protected DateFormat initialValue() {
       return new SimpleDateFormat("yyyy-MM-dd");
    }
};
```

说明：如果是 JDK8 的应用，可以使用 Instant 代替 Date，LocalDateTime 代替 Calendar，DateTimeFormatter 代替 SimpleDateFormat，官方给出的解释：simple beautiful strong immutable thread-safe。

6．【强制】高并发时，同步调用应该去考量锁的性能损耗。能用无锁数据结构，就不要用锁；能锁区块，就不要锁整个方法体；能用对象锁，就不要用类锁。

说明：尽可能使加锁的代码块工作量尽可能的小，避免在锁代码块中调用 RPC 方法。

7．【强制】对多个资源、数据库表、对象同时加锁时，需要保持一致的加锁顺序，否则可能会造成死锁。

说明：线程一需要对表 A、B、C 依次全部加锁后才可以进行更新操作，那么线程二的加锁顺序也必须是 A、B、C，否则可能出现死锁。

8．【强制】并发修改同一记录时，避免更新丢失，需要加锁。要么在应用层加锁，要么在缓存加锁，要么在数据库层使用乐观锁，使用 version 作为更新依据。

说明：如果每次访问冲突概率小于 20%，推荐使用乐观锁，否则使用悲观锁。乐观锁的重试次数不得小于 3 次。

9．【强制】多线程并行处理定时任务时，Timer 运行多个 TimeTask 时，只要其中之一没有捕获抛出的异常，其它任务便会自动终止运行，使用 ScheduledExecutorService 则没有这个问题。

10．【推荐】使用 CountDownLatch 进行异步转同步操作，每个线程退出前必须调用 countDown 方法，线程执行代码注意 catch 异常，确保 countDown 方法被执行到，避免主线程无法执行至 await 方法，直到超时才返回结果。

说明：注意，子线程抛出异常堆栈，不能在主线程 try-catch 到。

11．【推荐】避免 Random 实例被多线程使用，虽然共享该实例是线程安全的，但会因竞争同一 seed 导致的性能下降。

说明：Random 实例包括 java.util.Random 的实例或者 Math.random()的方式。

正例：在 JDK7 之后，可以直接使用 API ThreadLocalRandom，而在 JDK7 之前，需要编码保证每个线程持有一个实例。

12．【推荐】在并发场景下，通过双重检查锁（double-checked locking）实现延迟初始化的优化问题隐患(可参考 The "Double-Checked Locking is Broken" Declaration)，推荐解决方案中较为简单一种（适用于 JDK5 及以上版本），将目标属性声明为 volatile 型。

反例：

```
class Singleton {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) synchronized(this) {
            if (helper == null)
            helper = new Helper();
        }
        return helper;
    }
    // other methods and fields...
}
```

13．【参考】volatile 解决多线程内存不可见问题。对于一写多读，是可以解决变量同步问题，但是如果多写，同样无法解决线程安全问题。如果是 count++操作，使用如下类实现：AtomicInteger count = new AtomicInteger(); count.addAndGet(1); 如果是 JDK8，推荐使用 LongAdder 对象，比 AtomicLong 性能更好（减少乐观锁的重试次数）。

14．【参考】 HashMap 在容量不够进行 resize 时由于高并发可能出现死链，导致 CPU 飙升，在开发过程中可以使用其它数据结构或加锁来规避此风险。

15．【参考】ThreadLocal 无法解决共享对象的更新问题，ThreadLocal 对象建议使用 static 修饰。这个变量是针对一个线程内所有操作共享的，所以设置为静态变量，所有此类实例共享此静态变量 ，也就是说在类第一次被使用时装载，只分配一块存储空间，所有此类的对象(只要是这个线程内定义的)都可以操控这个变量。

（七）控制语句

1．【强制】在一个 switch 块内，每个 case 要么通过 break/return 等来终止，要么注释说明程序将继续执行到哪一个 case 为止；在一个 switch 块内，都必须包含一个 default 语句并且放在最后，即使它什么代码也没有。

2．【强制】在 if/else/for/while/do 语句中必须使用大括号。即使只有一行代码，避免采用单行的编码方式：if (condition) statements;

3．【推荐】表达异常的分支时，少用 if-else 方式，这种方式可以改写成：

```
if (condition) {
    ...
    return obj;
}
// 接着写 else 的业务逻辑代码;
```

说明：如果非得使用 if()...else if()...else...方式表达逻辑，【强制】避免后续代码维护困难，请勿超过 3 层。

正例：超过 3 层的 if-else 的逻辑判断代码可以使用卫语句、策略模式、状态模式等来实现，其中卫语句示例如下：

```
public void today() {
    if (isBusy()) {
        System.out.println(“change time.”);
        return;
    }

    if (isFree()) {
        System.out.println(“go to travel.”);
        return;
    }
```

```
        System.out.println("stay at home to learn Alibaba Java Coding Guidelines.");
        return;
    }
```

4．【推荐】除常用方法（如 getXxx/isXxx）等外，不要在条件判断中执行其它复杂的语句，将复杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。

说明：很多 if 语句内的逻辑相当复杂，阅读者需要分析条件表达式的最终结果，才能明确什么样的条件执行什么样的语句，那么，如果阅读者分析逻辑表达式错误呢？

正例：

```
// 伪代码如下
final boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed) {
    ...
}
```

反例：

```
if ((file.open(fileName, "w") != null) && (...) || (...)) {
    ...
}
```

5．【推荐】循环体中的语句要考量性能，以下操作尽量移至循环体外处理，如定义对象、变量、获取数据库连接，进行不必要的 try-catch 操作（这个 try-catch 是否可以移至循环体外）。

6．【推荐】接口入参保护，这种场景常见的是用于做批量操作的接口。

7．【参考】下列情形，需要进行参数校验：

1） 调用频次低的方法。

2） 执行时间开销很大的方法。此情形中，参数校验时间几乎可以忽略不计，但如果因为参数错误导致中间执行回退，或者错误，那得不偿失。

3） 需要极高稳定性和可用性的方法。

4） 对外提供的开放接口，不管是 RPC/API/HTTP 接口。

5） 敏感权限入口。

8．【参考】下列情形，不需要进行参数校验：

1） 极有可能被循环调用的方法。但在方法说明里必须注明外部参数检查要求。

2） 底层调用频度比较高的方法。毕竟是像纯净水过滤的最后一道，参数错误不太可能到底层才会暴露问题。一般 DAO 层与 Service 层都在同一个应用中，部署在同一台服务器中，所以 DAO 的参数校验，可以省略。

3） 被声明成 private 只会被自己代码所调用的方法，如果能够确定调用方法的代码传入参数已经做过检查或者肯定不会有问题，此时可以不校验参数。

（八）注释规约

1．【强制】类、类属性、类方法的注释必须使用 Javadoc 规范，使用/**内容*/格式，不得使用 // xxx 方式。

说明：在 IDE 编辑窗口中，Javadoc 方式会提示相关注释，生成 Javadoc 可以正确输出相应注释；在 IDE 中，工程调用方法时，不进入方法即可悬浮提示方法、参数、返回值的意义，提高阅读效率。

2．【强制】所有的抽象方法（包括接口中的方法）必须要用 Javadoc 注释、除了返回值、参数、异常说明外，还必须指出该方法做什么事情，实现什么功能。

说明：对子类的实现要求，或者调用注意事项，请一并说明。

3．【强制】所有的类都必须添加创建者和创建日期。

4．【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释使用/* */注释，注意与代码对齐。

5．【强制】所有的枚举类型字段必须要有注释，说明每个数据项的用途。

6．【推荐】与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词与关键字保持英文原文即可。

反例：“TCP 连接超时”解释成“传输控制协议连接超时”，理解反而费脑筋。

7．【推荐】代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑等的修改。

说明：代码与注释更新不同步，就像路网与导航软件更新不同步一样，如果导航软件严重滞后，就失去了导航的意义。

8．【参考】谨慎注释掉代码。在上方详细说明，而不是简单地注释掉。如果无用，则删除。

说明：代码被注释掉有两种可能性：1）后续会恢复此段代码逻辑。2）永久不用。前者如果没有备注信息，难以知晓注释动机。后者建议直接删掉（代码仓库保存了历史代码）。

9．【参考】对于注释的要求：第一、能够准确反应设计思想和代码逻辑；第二、能够描述业务含义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路；注释也是给继任者看的，使其能够快速接替自己的工作。

10．【参考】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

反例：

```
// put elephant into fridge
put(elephant, fridge);
```

方法名 put，加上两个有意义的变量名 elephant 和 fridge，已经说明了这是在干什么，语义清晰的代码不需要额外的注释。

11.【参考】特殊注释标记，请注明标记人与标记时间。注意及时处理这些标记，通过标记扫描，经常清理此类标记。线上故障有时候就是来源于这些标记处的代码。

1） 待办事宜（TODO）:（ 标记人，标记时间，[预计处理时间]）

表示需要实现，但目前还未实现的功能。这实际上是一个 Javadoc 的标签，目前的 Javadoc 还没有实现，但已经被广泛使用。只能应用于类，接口和方法（因为它是一个 Javadoc 标签）。

2） 错误，不能工作（FIXME）:（标记人，标记时间，[预计处理时间]）

在注释中用 FIXME 标记某代码是错误的，而且不能工作，需要及时纠正的情况。

（九）其它

1.【强制】在使用正则表达式时，利用好其预编译功能，可以有效加快正则匹配速度。

说明：不要在方法体内定义：Pattern pattern = Pattern.compile(规则);

2.【强制】velocity 调用 POJO 类的属性时，建议直接使用属性名取值即可，模板引擎会自动按规范调用 POJO 的 getXxx()，如果是 boolean 基本数据类型变量（boolean 命名不需要加 is 前缀），会自动调用 isXxx()方法。

说明：注意如果是 Boolean 包装类对象，优先调用 getXxx()的方法。

3.【强制】后台输送给页面的变量必须加$!{var}——中间的感叹号。

说明：如果 var=null 或者不存在，那么${var}会直接显示在页面上。

4.【强制】注意 Math.random() 这个方法返回是 double 类型，注意取值的范围 0≤x<1（能够取到零值，注意除零异常），如果想获取整数类型的随机数，不要将 x 放大 10 的若干倍然后取整，直接使用 Random 对象的 nextInt 或者 nextLong 方法。

5.【强制】获取当前毫秒数 System.currentTimeMillis(); 而不是 new Date().getTime();

说明：如果想获取更加精确的纳秒级时间值，使用 System.nanoTime()的方式。在 JDK8 中，针对统计时间等场景，推荐使用 Instant 类。

6.【推荐】不要在视图模板中加入任何复杂的逻辑。

说明：根据 MVC 理论，视图的职责是展示，不要抢模型和控制器的活。

7.【推荐】任何数据结构的构造或初始化，都应指定大小，避免数据结构无限增长吃光内存。

8.【推荐】及时清理不再使用的代码段或配置信息。

说明：对于垃圾代码或过时配置，坚决清理干净，避免程序过度臃肿，代码冗余。

正例：对于暂时被注释掉，后续可能恢复使用的代码片断，在注释代码上方，统一规定使用三个斜杠(///)来说明注释掉代码的理由。

二、异常日志

(一)异常处理

1. 【强制】Java 类库中定义的一类 RuntimeException 可以通过预先检查进行规避，而不应该通过 catch 来处理，比如：IndexOutOfBoundsException，NullPointerException 等等。
   说明：无法通过预检查的异常除外，如在解析一个外部传来的字符串形式数字时，通过 catch NumberFormatException 来实现。
   正例：`if (obj != null) {...}`
   反例：`try { obj.method() } catch (NullPointerException e) {...}`

2. 【强制】异常不要用来做流程控制，条件控制，因为异常的处理效率比条件分支低。

3. 【强制】对大段代码进行 try-catch，这是不负责任的表现。catch 时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的 catch 尽可能进行区分异常类型，再做对应的异常处理。

4. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。

5. 【强制】有 try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回滚事务。

6. 【强制】finally 块必须对资源对象、流对象进行关闭，有异常也要做 try-catch。
   说明：如果 JDK7 及以上，可以使用 try-with-resources 方式。

7. 【强制】不能在 finally 块中使用 return，finally 块中的 return 返回后方法结束执行，不会再执行 try 块中的 return 语句。

8. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。
   说明：如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。

9. 【推荐】方法的返回值可以为 null，不强制返回空集合，或者空对象等，必须添加注释充分说明什么情况下会返回 null 值。调用方需要进行 null 判断防止 NPE 问题。
   说明：本手册明确防止 NPE 是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回 null 的情况。

10. 【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景：
    1）返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE。
    反例：public int f() { return Integer 对象}， 如果为 null，自动解箱抛 NPE。

2） 数据库的查询结果可能为 null。

3） 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。

4） 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。

5） 对于 Session 中获取的数据，建议 NPE 检查，避免空指针。

6） 级联调用 obj.getA().getB().getC()；一连串调用，易产生 NPE。

正例：使用 JDK8 的 Optional 类来防止 NPE 问题。

11. 【推荐】定义时区分 unchecked / checked 异常，避免直接抛出 new RuntimeException()，更不允许抛出 Exception 或者 Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException 等。

12. 【参考】在代码中使用“抛异常”还是“返回错误码”，对于公司外的 http/api 开放接口必须使用“错误码”；而应用内部推荐异常抛出；跨应用间 RPC 调用优先考虑使用 Result 方式，封装 isSuccess()方法、“错误码”、“错误简短信息”。

说明：关于 RPC 方法返回方式使用 Result 方式的理由：

1）使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。

2）如果不加栈信息，只是 new 自定义异常，加入自己的理解的 error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。

13. 【参考】避免出现重复的代码（Don't Repeat Yourself），即 DRY 原则。

说明：随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是组件化。

正例：一个类中有多个 public 方法，都需要进行数行相同的参数校验操作，这个时候请抽取：

```
private boolean checkParam(DTO dto) {...}
```

（二）日志规约

1. 【强制】应用中不可直接使用日志系统（Log4j、Logback）中的 API，而应依赖使用日志框架 SLF4J 中的 API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger = LoggerFactory.getLogger(Abc.class);
```

2. 【强制】日志文件推荐至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。

3. 【强制】应用中的扩展日志（如打点、临时监控、访问日志等）命名方式：appName_logType_logName.log。logType:日志类型，推荐分类有 stats/desc/monitor/visit 等；logName:日志描述。这种命名的好处：通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。

正例：mppserver 应用中单独监控时区转换异常，如：

mppserver_monitor_timeZoneConvert.log

说明：推荐对日志进行分类，如将错误日志和业务日志分开存放，便于开发人员查看，也便于通过日志对系统进行及时监控。

4. 【强制】对 trace/debug/info 级别的日志输出，必须使用条件输出形式或者使用占位符的方式。

说明：logger.debug("Processing trade with id: " + id + " and symbol: " + symbol); 如果日志级别是 warn，上述日志不会打印，但是会执行字符串拼接操作，如果 symbol 是对象，会执行 toString()方法，浪费了系统资源，执行了上述操作，最终日志却没有打印。

正例：（条件）

```
if (logger.isDebugEnabled()) {
    logger.debug("Processing trade with id: " + id + " and symbol: " + symbol);
}
```

正例：（占位符）

```
logger.debug("Processing trade with id: {} and symbol : {} ", id, symbol);
```

5. 【强制】避免重复打印日志，浪费磁盘空间，务必在 log4j.xml 中设置 additivity=false。

正例：<logger name="com.taobao.dubbo.config" additivity="false">

6. 【强制】异常信息应该包括两类信息：案发现场信息和异常堆栈信息。如果不处理，那么通过关键字 throws 往上抛出。

正例：logger.error(各类参数或者对象 toString + "_" + e.getMessage(), e);

7. 【推荐】谨慎地记录日志。生产环境禁止输出 debug 日志；有选择地输出 info 日志；如果使用 warn 来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘撑爆，并记得及时删除这些观察日志。

说明：大量地输出无效日志，不利于系统性能提升，也不利于快速定位错误点。记录日志时请思考：这些日志真的有人看吗？看到这条日志你能做什么？能不能给问题排查带来好处？

8. 【参考】可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适从。注意日志输出的级别，error 级别只记录系统逻辑出错、异常等重要的错误信息。如非必要，请不要在此场景打出 error 级别。

三、单元测试

1. 【强制】好的单元测试必须遵守 AIR 原则。
   说明：单元测试在线上运行时，感觉像空气（AIR）一样并不存在，但在测试质量的保障上，却是非常关键的。好的单元测试宏观上来说，具有自动化、独立性、可重复执行的特点。
   - A：Automatic（自动化）
   - I：Independent（独立性）
   - R：Repeatable（可重复）
2. 【强制】单元测试应该是全自动执行的，并且非交互式的。测试框架通常是定期执行的，执行过程必须完全自动化才有意义。输出结果需要人工检查的测试不是一个好的单元测试。单元测试中不准使用 System.out 来进行人肉验证，必须使用 assert 来验证。
3. 【强制】保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间决不能互相调用，也不能依赖执行的先后次序。
   反例：method2 需要依赖 method1 的执行，将执行结果做为 method2 的输入。
4. 【强制】单元测试是可以重复执行的，不能受到外界环境的影响。
   说明：单元测试通常会被放到持续集成中，每次有代码 check in 时单元测试都会被执行。如果单测对外部环境（网络、服务、中间件等）有依赖，容易导致持续集成机制的不可用。
   正例：为了不受外界环境影响，要求设计代码时就把 SUT 的依赖改成注入，在测试时用 spring 这样的 DI 框架注入一个本地（内存）实现或者 Mock 实现。
5. 【强制】对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级别，一般是方法级别。
   说明：只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检查跨类或者跨系统的交互逻辑，那是集成测试的领域。
6. 【强制】核心业务、核心应用、核心模块的增量代码确保单元测试通过。
   说明：新增代码及时补充单元测试，如果新增代码影响了原有单元测试，请及时修正。
7. 【强制】单元测试代码必须写在如下工程目录：src/test/java，不允许写在业务代码目录下。
   说明：源码构建时会跳过此目录，而单元测试框架默认是扫描此目录。
8. 【推荐】单元测试的基本目标：语句覆盖率达到 70%；核心模块的语句覆盖率和分支覆盖率都要达到 100%
   说明：在工程规约的应用分层中提到的 DAO 层，Manager 层，可重用度高的 Service，都应该进行单元测试。

9．【推荐】编写单元测试代码遵守 BCDE 原则，以保证被测试模块的交付质量。

- B：Border，边界值测试，包括循环边界、特殊取值、特殊时间点、数据顺序等。
- C：Correct，正确的输入，并得到预期的结果。
- D：Design，与设计文档相结合，来编写单元测试。
- E：Error，强制错误信息输入（如：非法数据、异常流程、非业务允许输入等），并得到预期的结果。

10．【推荐】对于数据库相关的查询，更新，删除等操作，不能假设数据库里的数据是存在的，或者直接操作数据库把数据插入进去，请使用程序插入或者导入数据的方式来准备数据。

反例：删除某一行数据的单元测试，在数据库中，先直接手动增加一行作为删除目标，但是这一行新增数据并不符合业务插入规则，导致测试结果异常。

11．【推荐】和数据库相关的单元测试，可以设定自动回滚机制，不给数据库造成脏数据。或者对单元测试产生的数据有明确的前后缀标识。

正例：在 RDC 内部单元测试中，使用 RDC_UNIT_TEST_的前缀标识数据。

12．【推荐】对于不可测的代码建议做必要的重构，使代码变得可测，避免为了达到测试要求而书写不规范测试代码。

13．【推荐】在设计评审阶段，开发人员需要和测试人员一起确定单元测试范围，单元测试最好覆盖所有测试用例（UC）。

14．【推荐】单元测试作为一种质量保障手段，不建议项目发布后补充单元测试用例，建议在项目提测前完成单元测试。

15．【参考】为了更方便地进行单元测试，业务代码应避免以下情况：

- 构造方法中做的事情过多。
- 存在过多的全局变量和静态方法。
- 存在过多的外部依赖。
- 存在过多的条件语句。

  说明：多层条件语句建议使用卫语句、策略模式、状态模式等方式重构。

16．【参考】不要对单元测试存在如下误解：

- 那是测试同学干的事情。本文是开发手册，凡是本文内容都是与开发同学强相关的。
- 单元测试代码是多余的。汽车的整体功能与各单元部件的测试正常与否是强相关的。
- 单元测试代码不需要维护。一年半载后，那么单元测试几乎处于废弃状态。
- 单元测试与线上故障没有辩证关系。好的单元测试能够最大限度地规避线上故障。

四、安全规约

1. 【强制】隶属于用户个人的页面或者功能必须进行权限控制校验。
   说明：防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信内容、修改他人的订单。

2. 【强制】用户敏感数据禁止直接展示，必须对展示数据进行脱敏。
   说明：查看个人手机号码会显示成:158****9119，隐藏中间 4 位，防止隐私泄露。

3. 【强制】用户输入的 SQL 参数严格使用参数绑定或者 METADATA 字段值限定，防止 SQL 注入，禁止字符串拼接 SQL 访问数据库。

4. 【强制】用户请求传入的任何参数必须做有效性验证。
   说明：忽略参数校验可能导致：

   - page size 过大导致内存溢出
   - 恶意 order by 导致数据库慢查询
   - 任意重定向
   - SQL 注入
   - 反序列化注入
   - 正则输入源串拒绝服务 ReDoS

   说明：Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

5. 【强制】禁止向 HTML 页面输出未经安全过滤或未正确转义的用户数据。

6. 【强制】表单、AJAX 提交必须执行 CSRF 安全过滤。
   说明：CSRF(Cross-site request forgery)跨站请求伪造是一类常见编程漏洞。对于存在 CSRF 漏洞的应用/网站，攻击者可以事先构造好 URL，只要受害者用户一访问，后台便在用户不知情情况下对数据库中用户参数进行相应修改。

7. 【强制】在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放限制，如数量限制、疲劳度控制、验证码校验，避免被滥刷、资损。
   说明：如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其它用户，并造成短信平台资源浪费。

8. 【推荐】发贴、评论、发送即时消息等用户生成内容的场景必须实现防刷、文本内容违禁词过滤等风控策略。

五、MySQL 数据库

（一）建表规约

1. 【强制】表达是与否概念的字段，必须使用 is_xxx 的方式命名，数据类型是 unsigned tinyint（ 1 表示是，0 表示否）。
   说明：任何字段如果为非负数，必须是 unsigned。
   正例：表达逻辑删除的字段名 is_deleted，1 表示删除，0 表示未删除。

2. 【强制】表名、字段名必须使用小写字母或数字，禁止出现数字开头，禁止两个下划线中间只出现数字。数据库字段名的修改代价很大，因为无法进行预发布，所以字段名称需要慎重考虑。
   说明：MySQL 在 Windows 下不区分大小写，但在 Linux 下默认是区分大小写。因此，数据库名、表名、字段名，都不允许出现任何大写字母，避免节外生枝。
   正例：aliyun_admin，rdc_config，level3_name
   反例：AliyunAdmin，rdcConfig，level_3_name

3. 【强制】表名不使用复数名词。
   说明：表名应该仅仅表示表里面的实体内容，不应该表示实体数量，对应于 DO 类名也是单数形式，符合表达习惯。

4. 【强制】禁用保留字，如 desc、range、match、delayed 等，请参考 MySQL 官方保留字。

5. 【强制】主键索引名为 pk_字段名；唯一索引名为 uk_字段名；普通索引名则为 idx_字段名。
   说明：pk_ 即 primary key；uk_ 即 unique key；idx_ 即 index 的简称。

6. 【强制】小数类型为 decimal，禁止使用 float 和 double。
   说明：float 和 double 在存储的时候，存在精度损失的问题，很可能在值的比较时，得到不正确的结果。如果存储的数据范围超过 decimal 的范围，建议将数据拆成整数和小数分开存储。

7. 【强制】如果存储的字符串长度几乎相等，使用 char 定长字符串类型。

8. 【强制】varchar 是可变长字符串，不预先分配存储空间，长度不要超过 5000，如果存储长度大于此值，定义字段类型为 text，独立出来一张表，用主键来对应，避免影响其它字段索引效率。

9. 【强制】表必备三字段：id，gmt_create，gmt_modified。
   说明：其中 id 必为主键，类型为 unsigned bigint、单表时自增、步长为 1。gmt_create，gmt_modified 的类型均为 date_time 类型，前者现在时表示主动创建，后者过去分词表示被动更新。

10. 【推荐】表的命名最好是加上“业务名称_表的作用”。
    正例：alipay_task / force_project / trade_config

11.【推荐】库名与应用名称尽量一致。

12.【推荐】如果修改字段含义或对字段表示的状态追加时，需要及时更新字段注释。

13.【推荐】字段允许适当冗余，以提高查询性能，但必须考虑数据一致。冗余字段应遵循：

1）不是频繁修改的字段。

2）不是 varchar 超长字段，更不能是 text 字段。

正例：商品类目名称使用频率高，字段长度短，名称基本一成不变，可在相关联的表中冗余存储类目名称，避免关联查询。

14.【推荐】单表行数超过 500 万行或者单表容量超过 2GB，才推荐进行分库分表。

说明：如果预计三年后的数据量根本达不到这个级别，请不要在创建表时就分库分表。

15.【参考】合适的字符存储长度，不但节约数据库表空间、节约索引存储，更重要的是提升检索速度。

正例：如下表，其中无符号值可以避免误存负数，且扩大了表示范围。

对象	年龄区间	类型	字节	表示范围
人	150 岁之内	unsigned tinyint	1	无符号值：0 到 255
龟	数百岁	unsigned smallint	2	无符号值：0 到 65535
恐龙化石	数千万年	unsigned int	4	无符号值：0 到约 42.9 亿
太阳	约 50 亿年	unsigned bigint	8	无符号值：0 到约 10 的 19 次方

（二）索引规约

1.【强制】业务上具有唯一特性的字段，即使是多个字段的组合，也必须建成唯一索引。

说明：不要以为唯一索引影响了 insert 速度，这个速度损耗可以忽略，但提高查找速度是明显的；另外，即使在应用层做了非常完善的校验控制，只要没有唯一索引，根据墨菲定律，必然有脏数据产生。

2.【强制】超过三个表禁止 join。需要 join 的字段，数据类型必须绝对一致；多表关联查询时，保证被关联的字段需要有索引。

说明：即使双表 join 也要注意表索引、SQL 性能。

3.【强制】在 varchar 字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据实际文本区分度决定索引长度即可。

说明：索引的长度与区分度是一对矛盾体，一般对字符串类型数据，长度为 20 的索引，区分

度会高达 90%以上，可以使用 count(distinct left(列名，索引长度))/count(*)的区分度来确定。

4．【强制】页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决。

说明：索引文件具有 B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引。

5．【推荐】如果有 order by 的场景，请注意利用索引的有序性。order by 最后的字段是组合索引的一部分，并且放在索引组合顺序的最后，避免出现 file_sort 的情况，影响查询性能。

正例：where a=? and b=? order by c; 索引：a_b_c

反例：索引中有范围查找，那么索引有序性无法利用，如：WHERE a>10 ORDER BY b; 索引 a_b 无法排序。

6．【推荐】利用覆盖索引来进行查询操作，避免回表。

说明：如果一本书需要知道第 11 章是什么标题，会翻开第 11 章对应的那一页吗？目录浏览一下就好，这个目录就是起到覆盖索引的作用。

正例：能够建立索引的种类：主键索引、唯一索引、普通索引，而覆盖索引是一种查询的一种效果，用 explain 的结果，extra 列会出现：using index。

7．【推荐】利用延迟关联或者子查询优化超多分页场景。

说明：MySQL 并不是跳过 offset 行，而是取 offset+N 行，然后返回放弃前 offset 行，返回 N 行，那当 offset 特别大的时候，效率就非常的低下，要么控制返回的总页数，要么对超过特定阈值的页数进行 SQL 改写。

正例：先快速定位需要获取的 id 段，然后再关联：

```
SELECT a.* FROM 表 1 a, (select id from 表 1 where 条件 LIMIT 100000,20 ) b where a.id=b.id
```

8．【推荐】SQL 性能优化的目标：至少要达到 range 级别，要求是 ref 级别，如果可以是 consts 最好。

说明：

1）consts 单表中最多只有一个匹配行（主键或者唯一索引），在优化阶段即可读取到数据。

2）ref 指的是使用普通的索引（normal index）。

3）range 对索引进行范围检索。

反例：explain 表的结果，type=index，索引物理文件全扫描，速度非常慢，这个 index 级别比较 range 还低，与全表扫描是小巫见大巫。

9．【推荐】建组合索引的时候，区分度最高的在最左边。

正例：如果 where a=? and b=? ，a 列的几乎接近于唯一值，那么只需要单建 idx_a 索引即可。

说明：存在非等号和等号混合判断条件时，在建索引时，请把等号条件的列前置。如：where a>? and b=? 那么即使 a 的区分度更高，也必须把 b 放在索引的最前列。

10.【推荐】防止因字段类型不同造成的隐式转换，导致索引失效。

11.【参考】创建索引时避免有如下极端误解：

1）宁滥勿缺。认为一个查询就需要建一个索引。

2）宁缺勿滥。认为索引会消耗空间、严重拖慢更新和新增速度。

3）抵制惟一索引。认为业务的惟一性一律需要在应用层通过“先查后插”方式解决。

（三）SQL 语句

1.【强制】不要使用 count(列名)或 count(常量)来替代 count(*)，count(*)是 SQL92 定义的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。

说明：count(*)会统计值为 NULL 的行，而 count(列名)不会统计此列为 NULL 值的行。

2.【强制】count(distinct col) 计算该列除 NULL 之外的不重复行数，注意 count(distinct col1, col2) 如果其中一列全为 NULL，那么即使另一列有不同的值，也返回为 0。

3.【强制】当某一列的值全是 NULL 时，count(col)的返回结果为 0，但 sum(col)的返回结果为 NULL，因此使用 sum()时需注意 NPE 问题。

正例：可以使用如下方式来避免 sum 的 NPE 问题：SELECT IF(ISNULL(SUM(g)),0,SUM(g)) FROM table;

4.【强制】使用 ISNULL()来判断是否为 NULL 值。

说明：NULL 与任何值的直接比较都为 NULL。

1） NULL<>NULL 的返回结果是 NULL，而不是 false。

2） NULL=NULL 的返回结果是 NULL，而不是 true。

3） NULL<>1 的返回结果是 NULL，而不是 true。

5.【强制】在代码中写分页查询逻辑时，若 count 为 0 应直接返回，避免执行后面的分页语句。

6.【强制】不得使用外键与级联，一切外键概念必须在应用层解决。

说明：以学生和成绩的关系为例，学生表中的 student_id 是主键，那么成绩表中的 student_id 则为外键。如果更新学生表中的 student_id，同时触发成绩表中的 student_id 更新，即为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

7.【强制】禁止使用存储过程，存储过程难以调试和扩展，更没有移植性。

8.【强制】数据订正时，删除和修改记录时，要先 select，避免出现误删除，确认无误才能执行更新语句。

9．【推荐】in 操作能避免则避免，若实在避免不了，需要仔细评估 in 后边的集合元素数量，控制在 1000 个之内。

10．【参考】如果有全球化需要，所有的字符存储与表示，均以 utf-8 编码，注意字符统计函数的区别。

说明：

SELECT LENGTH("轻松工作"); 返回为 12

SELECT CHARACTER_LENGTH("轻松工作"); 返回为 4

如果需要存储表情，那么选择 utfmb4 来进行存储，注意它与 utf-8 编码的区别。

11．【参考】TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少，但 TRUNCATE 无事务且不触发 trigger，有可能造成事故，故不建议在开发代码中使用此语句。

说明：TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同。

（四）ORM 映射

1．【强制】在表查询中，一律不要使用 * 作为查询的字段列表，需要哪些字段必须明确写明。

说明：1）增加查询分析器解析成本。2）增减字段容易与 resultMap 配置不一致。

2．【强制】POJO 类的布尔属性不能加 is，而数据库字段必须加 is_，要求在 resultMap 中进行字段与属性之间的映射。

说明：参见定义 POJO 类以及数据库字段定义规定，在<resultMap>中增加映射，是必须的。在 MyBatis Generator 生成的代码中，需要进行对应的修改。

3．【强制】不要用 resultClass 当返回参数，即使所有类属性名与数据库字段一一对应，也需要定义；反过来，每一个表也必然有一个与之对应。

说明：配置映射关系，使字段与 DO 类解耦，方便维护。

4．【强制】sql.xml 配置参数使用：#{}，#param# 不要使用${} 此种方式容易出现 SQL 注入。

5．【强制】iBATIS 自带的 queryForList(String statementName,int start,int size)不推荐使用。

说明：其实现方式是在数据库取到 statementName 对应的 SQL 语句的所有记录，再通过 subList 取 start,size 的子集合。

正例：
```
Map<String, Object> map = new HashMap<String, Object>();
map.put("start", start);
map.put("size", size);
```

6．【强制】不允许直接拿 HashMap 与 Hashtable 作为查询结果集的输出。

说明：resultClass="Hashtable"，会置入字段名和属性值，但是值的类型不可控。

7．【强制】更新数据表记录时，必须同时更新记录对应的 gmt_modified 字段值为当前时间。

8．【推荐】不要写一个大而全的数据更新接口。传入为 POJO 类，不管是不是自己的目标更新字段，都进行 update table set c1=value1,c2=value2,c3=value3; 这是不对的。执行 SQL 时，不要更新无改动的字段，一是易出错；二是效率低；三是增加 binlog 存储。

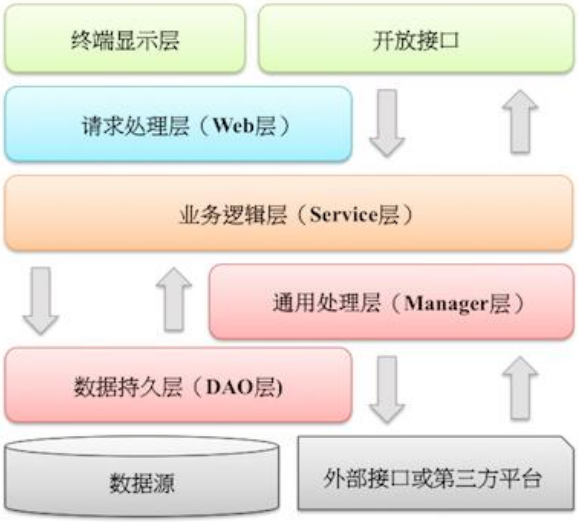
9．【参考】@Transactional 事务不要滥用。事务会影响数据库的 QPS，另外使用事务的地方需要考虑各方面的回滚方案，包括缓存回滚、搜索引擎回滚、消息补偿、统计修正等。

10．【参考】<isEqual>中的 compareValue 是与属性值对比的常量，一般是数字，表示相等时带上此条件；<isNotEmpty>表示不为空且不为 null 时执行；<isNotNull>表示不为 null 值时执行。

六、工程结构

(一)应用分层

1．【推荐】图中默认上层依赖于下层，箭头关系表示可直接依赖，如：开放接口层可以依赖于 Web 层，也可以直接依赖于 Service 层，依此类推：



- **开放接口层**：可直接封装 Service 方法暴露成 RPC 接口；通过 Web 封装成 http 接口；进行网关安全控制、流量控制等。
- **终端显示层**：各个端的模板渲染并执行显示的层。当前主要是 velocity 渲染，JS 渲染，JSP 渲染，移动端展示等。
- **Web 层**：主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。
- **Service 层**：相对具体的业务逻辑服务层。
- **Manager 层**：通用业务处理层，它有如下特征：
  1） 对第三方平台封装的层，预处理返回结果及转化异常信息；
  2） 对 Service 层通用能力的下沉，如缓存方案、中间件通用处理；
  3） 与 DAO 层交互，对多个 DAO 的组合复用。
- **DAO 层**：数据访问层，与底层 MySQL、Oracle、Hbase 等进行数据交互。
- **外部接口或第三方平台**：包括其它部门 RPC 开放接口，基础平台，其它公司的 HTTP 接口。

2．【参考】 （分层异常处理规约）在 DAO 层，产生的异常类型有很多，无法用细粒度的异常进行 catch，使用 catch(Exception e)方式，并 throw new DAOException(e)，不需要打印日志，因为日志在 Manager/Service 层一定需要捕获并打到日志文件中去，如果同台服务器再打日志，浪费性能和存储。在 Service 层出现异常时，必须记录出错日志到磁盘，尽可能带上参数信息，相当于保护案发现场。如果 Manager 层与 Service 同机部署，日志方式与 DAO 层处理一致，如果是单独部署，则采用与 Service 一致的处理方式。Web 层绝不应该继续往上

抛异常，因为已经处于顶层，如果意识到这个异常将导致页面无法正常渲染，那么就应该直接跳转到友好错误页面，加上用户容易理解的错误提示信息。开放接口层要将异常处理成错误码和错误信息方式返回。

3．【参考】分层领域模型规约：

- DO（Data Object）：与数据库表结构一一对应，通过 DAO 层向上传输数据源对象。
- DTO（Data Transfer Object）：数据传输对象，Service 或 Manager 向外传输的对象。
- BO（Business Object）：业务对象。由 Service 层输出的封装业务逻辑的对象。
- AO（Application Object）：应用对象。在 Web 层与 Service 层之间抽象的复用对象模型，极为贴近展示层，复用度不高。
- VO（View Object）：显示层对象，通常是 Web 向模板渲染引擎层传输的对象。
- Query：数据查询对象，各层接收上层的查询请求。注意超过 2 个参数的查询封装，禁止使用 Map 类来传输。

（二）二方库依赖

1．【强制】定义 GAV 遵从以下规则：

1） GroupID 格式：com.{公司/BU }.业务线.[子业务线]，最多 4 级。

说明：{公司/BU} 例如：alibaba/taobao/tmall/aliexpress 等 BU 一级；子业务线可选。

正例：com.taobao.jstorm 或 com.alibaba.dubbo.register

2） ArtifactID 格式：产品线名-模块名。语义不重复不遗漏，先到中央仓库去查证一下。

正例：dubbo-client / fastjson-api / jstorm-tool

3） Version：详细规定参考下方。

2．【强制】二方库版本号命名方式：主版本号.次版本号.修订号

1） **主版本号**：产品方向改变，或者大规模 API 不兼容，或者架构不兼容升级。

2） **次版本号**：保持相对兼容性，增加主要功能特性，影响范围极小的 API 不兼容修改。

3） **修订号**：保持完全兼容性，修复 BUG、新增次要功能特性等。

说明：注意起始版本号必须为：1.0.0，而不是 0.0.1　正式发布的类库必须先去中央仓库进行查证，使版本号有延续性，正式版本号不允许覆盖升级。如当前版本：1.3.3，那么下一个合理的版本号：1.3.4 或 1.4.0 或 2.0.0

3．【强制】线上应用不要依赖 SNAPSHOT 版本（安全包除外）。

说明：不依赖 SNAPSHOT 版本是保证应用发布的幂等性。另外，也可以加快编译时的打包构建。

4. 【强制】二方库的新增或升级，保持除功能点之外的其它 jar 包仲裁结果不变。如果有改变，必须明确评估和验证，建议进行 dependency:resolve 前后信息比对，如果仲裁结果完全不一致，那么通过 dependency:tree 命令，找出差异点，进行<excludes>排除 jar 包。

5. 【强制】二方库里可以定义枚举类型，参数可以使用枚举类型，但是接口返回值不允许使用枚举类型或者包含枚举类型的 POJO 对象。

6. 【强制】依赖于一个二方库群时，必须定义一个统一的版本变量，避免版本号不一致。
   说明：依赖 springframework-core,-context,-beans，它们都是同一个版本，可以定义一个变量来保存版本：${spring.version}，定义依赖的时候，引用该版本。

7. 【强制】禁止在子项目的 pom 依赖中出现相同的 GroupId，相同的 ArtifactId，但是不同的 Version。
   说明：在本地调试时会使用各子项目指定的版本号，但是合并成一个 war，只能有一个版本号出现在最后的 lib 目录中。可能出现线下调试是正确的，发布到线上却出故障的问题。

8. 【推荐】所有 pom 文件中的依赖声明放在<dependencies>语句块中，所有版本仲裁放在<dependencyManagement>语句块中。
   说明：<dependencyManagement>里只是声明版本，并不实现引入，因此子项目需要显式的声明依赖，version 和 scope 都读取自父 pom。而<dependencies>所有声明在主 pom 的<dependencies>里的依赖都会自动引入，并默认被所有的子项目继承。

9. 【推荐】二方库不要有配置项，最低限度不要再增加配置项。

10. 【参考】为避免应用二方库的依赖冲突问题，二方库发布者应当遵循以下原则：
    1）**精简可控原则**。移除一切不必要的 API 和依赖，只包含 Service API、必要的领域模型对象、Utils 类、常量、枚举等。如果依赖其它二方库，尽量是 provided 引入，让二方库使用者去依赖具体版本号；无 log 具体实现，只依赖日志框架。
    2）**稳定可追溯原则**。每个版本的变化应该被记录，二方库由谁维护，源码在哪里，都需要能方便查到。除非用户主动升级版本，否则公共二方库的行为不应该发生变化。

（三）服务器

1. 【推荐】高并发服务器建议调小 TCP 协议的 time_wait 超时时间。
   说明：操作系统默认 240 秒后，才会关闭处于 time_wait 状态的连接，在高并发访问下，服务器端会因为处于 time_wait 的连接数太多，可能无法建立新的连接，所以需要在服务器上调小此等待值。
   正例：在 linux 服务器上请通过变更/etc/sysctl.conf 文件去修改该缺省值（秒）：

```
net.ipv4.tcp_fin_timeout = 30
```

2．【推荐】调大服务器所支持的最大文件句柄数（File Descriptor，简写为 fd）。

说明：主流操作系统的设计是将 TCP/UDP 连接采用与文件一样的方式去管理，即一个连接对应于一个 fd。主流的 linux 服务器默认所支持最大 fd 数量为 1024，当并发连接数很大时很容易因为 fd 不足而出现“open too many files”错误，导致新的连接无法建立。 建议将 linux 服务器所支持的最大句柄数调高数倍（与服务器的内存数量相关）。

3．【推荐】给 JVM 设置-XX:+HeapDumpOnOutOfMemoryError 参数，让 JVM 碰到 OOM 场景时输出 dump 信息。

说明：OOM 的发生是有概率的，甚至有规律地相隔数月才出现一例，出现时的现场信息对查错非常有价值。

4．【推荐】在线上生产环境，JVM 的 Xms 和 Xmx 设置一样大小的内存容量，避免在 GC 后调整堆大小带来的压力。

5．【参考】服务器内部重定向使用 forward；外部重定向地址使用 URL 拼装工具类来生成，否则会带来 URL 维护不一致的问题和潜在的安全风险。

附 1：版本历史

版本号	更新日期	备注
1.0.0	2017.2.9	阿里巴巴集团正式对外发布
1.0.1	2017.2.13	1）修正 String[]的前后矛盾。2）vm 修正成 velocity。3）修正 countdown 描述错误。
1.0.2	2017.2.20	1）去除文底水印。2）数据类型中引用太阳系年龄问题。3）修正关于异常和方法签名的部分描述。4）修正 final 描述。5）去除 Comparator 部分描述。
1.1.0	2017.2.27	1）增加前言。2）增加<? extends T>描述和说明。3）增加版本历史。4）增加专有名词解释。
1.1.1	2017.3.31	修正页码总数和部分示例。
1.2.0	2017.5.20	1）根据云栖社区的“聚能聊”活动反馈，对手册的页码、排版、描述进行修正。2）增加 final 的适用场景描述。3）增加关于锁的粒度的说明。4）增加“指定集合大小”的详细说明以及正反例。5）增加卫语句的示例代码。6）明确数据库表示删除概念的字段名为 is_deleted
1.3.0	2017.9.25	增加单元测试规约（PDF 终极版），阿里开源的 IDE 代码规约检测插件：点此下载 更多及时信息，请关注《阿里巴巴 Java 开发手册》官方公众号：

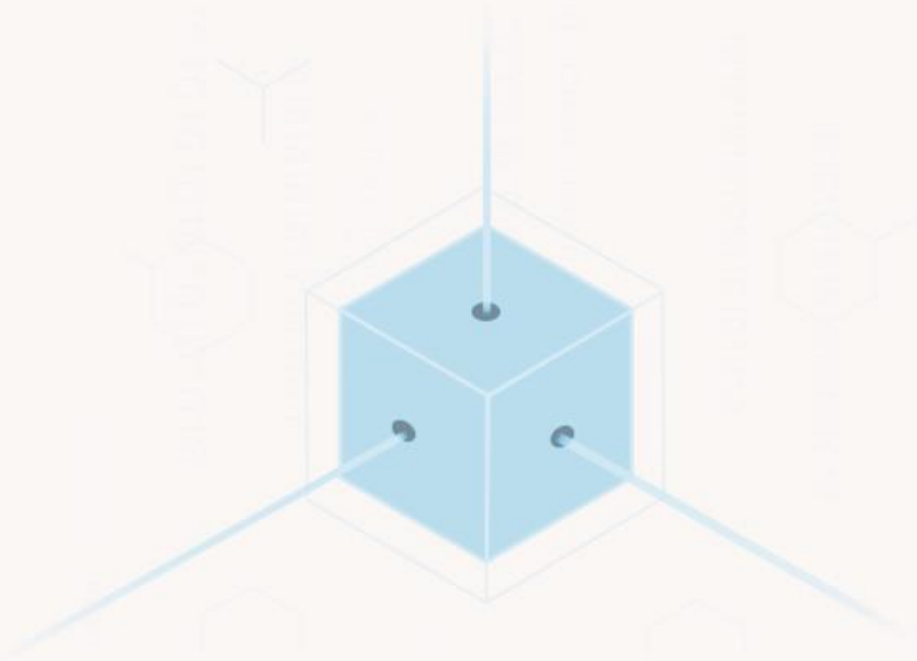
附 2：本手册专有名词

1. **POJO**（Plain Ordinary Java Object）：在本手册中，POJO 专指只有 setter / getter / toString 的简单类，包括 DO/DTO/BO/VO 等。
2. **GAV**（GroupId、ArtifactctId、Version）：Maven 坐标，是用来唯一标识 jar 包。
3. **OOP**（Object Oriented Programming）：本手册泛指类、对象的编程处理方式。
4. **ORM**（Object Relation Mapping）：对象关系映射，对象领域模型与底层数据之间的转换，本文泛指 iBATIS，mybatis 等框架。
5. **NPE**（java.lang.NullPointerException）：空指针异常。
6. **SOA**（Service-Oriented Architecture）：面向服务架构，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用，有利于提升组件可重用性，可维护性。
7. **一方库**：本工程内部子项目模块依赖的库（jar 包）。
8. **二方库**：公司内部发布到中央仓库，可供公司内部其它应用依赖的库（jar 包）。
9. **三方库**：公司之外的开源库（jar 包）。
10. **IDE**（Integrated Development Environment）：用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面等工具，本《手册》泛指 IntelliJ IDEA 和 eclipse。

法律声明

本手册为阿里巴巴集团技术部的技术分享，版权归阿里巴巴集团所有，仅供大家交流、学习及研究使用，禁止用于商业用途，违者必究。

Alibaba Group
阿里巴巴集团



关注阿里技术

访问阿里云栖社区